UNIWERSYTET WSB MERITO W GDAŃSKU

WYDZIAŁ INFORMATYKI I NOWYCH   
TECHNOLOGII

**ZASTOSOWANIE SIECI NEURONOWYCH W ROZPOZNAWANIU RAKA SKÓRY NA PODSTAWIE ANALIZY OBRAZÓW DERMATOSKOPOWYCH**

Praca inżynierska

na kierunku Informatyka

Praca napisana pod kierunkiem

dr inż. Pawła Tomkiewicza

Marek Małek, 56572, Gdańsk 2024

**Spis treści**

[WSTĘP 2](#_Toc124960765)

[Rozdział 1 4](#_Toc124960766)

[1.1 Wprowadzenie 4](#_Toc124960767)

[1.2 C# i .Net 4](#_Toc124960768)

[Rozdział 2 6](#_Toc124960769)

[2.1 Rider 6](#_Toc124960771)

[2.2 DataGrip 6](#_Toc124960772)

[2.3 Visual Studio Code 6](#_Toc124960773)

[2.4 Apache Cassandra 6](#_Toc124960774)

[2.5 ElasticSearch 7](#_Toc124960775)

[2.6 Kafka 7](#_Toc124960776)

[Rozdział 3 8](#_Toc124960777)

[3.1 Struktura aplikacji 8](#_Toc124960779)

[3.2 Wymagania techniczne 8](#_Toc124960780)

[Rozdział 4 9](#_Toc124960781)

[4.1 Implementacja 9](#_Toc124960783)

[4.2 Testowanie 14](#_Toc124960784)

[4.3 Wdrożenie 15](#_Toc124960785)

[BIBLIOGRAFIA 16](#_Toc124960786)

[Opracowania książkowe 16](#_Toc124960787)

[Netografia 16](#_Toc124960788)

[SPIS RYSUNKÓW 17](#_Toc124960789)

# WSTĘP

Celem pracy jest wykonanie prototypu produktu potrafiącego na podstawie zdjęć wykonanych np. przy pomocy smartfona rozpoznawać raka skóry. W skład prototypu wchodzić będą następujące części:

* Aplikacja mobilna napisana w języku Dart przy użyciu biblioteki Flutter; chociaż aplikacje pisane przy użyciu tych technologii można uruchamiać na dowolnym smartfonie, elementem prototypu będzie jedynie aplikacja uruchamiana w systemie Android
* Aplikacja serwerowa napisana w języku Python przy użyciu biblioteki Flask, a uruchamiana w kontenerach Docker, w chmurze Azure; jej elementem będzie wytrenowana, docelowa sieć neuronowa
* Zestaw notatników Jupyter zawierających efekty eksperymentów polegających na trenowaniu kilku wariantów architektury InceptionResNetV2 na bazie biblioteki TensorFlow oraz innych, pomocniczych sieci neuronowych

Poza wymienionymi składowymi, wykonane zostaną także skrypty ułatwiające zarządzanie kodem, deploymentem i uruchamianiem aplikacji. Będą to programy Powershell a także niezbędne skrypty Docker.

Odbiorcami dostarczonych funkcjonalności mogą być zarówno indywidualni użytkownicy, jak i lekarze pierwszego kontaktu, którzy dzięki pomocy opisywanego produktu będą zdolni wspomóc swój proces decyzyjny. Jako, że jest to prototyp demonstrujący co w dziedzinie wykrywania raka skóry można osiągnąć przy pomocy sieci neuronowych, produkt ten będzie także stanowił punkt wyjściowy rozważań na temat ewentualnych usprawnień, które można byłoby poczynić, aby jeszcze bardziej podnieść jego skuteczność.

W rozdziale pierwszym i drugim została zawarta część teoretyczna pracy. Opisano w nich matematyczną podbudowę sieci neuronowych, w szczególności splotowych sieci neuronowych (CNN), a także przybliżono w nich informacje dotyczące technologii oraz narzędzi potrzebnych do stworzenia rdzenia – sieci neuronowej – aplikacji „Skin Cancer Detector”.

W rozdziale trzecim przedstawiono architektury i opisano kolejne sieci neuronowe używane na różnych etapach eksperymentów mających na celu wytrenowanie jak najdokładniejszego klasyfikatora. Opisano problemy, z którymi spotkano się w trakcie prac.

Rozdział czwarty poświęcono na omówienie części oprogramowania widocznej na zewnątrz, a więc aplikacji mobilnej oraz serwerowej. Umieszczono tu także informacje dotyczące oprogramowania Docker i chmury Azure oraz ich istotności w wytwarzaniu tego typu oprogramowania.

W ostatnim rozdziale zawarto najważniejsze informacje odnoszące się do funkcjonalności aplikacji. Przedstawiono przebieg jej tworzenia – implementację poszczególnych, użytych finalnie sieci neuronowych, a także tych, które nie zostały użyte, ale przyczyniły się do znalezienia właściwych architektur sieci. Dalsza część poświęcona została omówieniu procesu wytwarzania aplikacji mobilnych i serwerowej. Pracę kończy synteza rozważań i dokonań, czyli zakończenie.

# Rozdział 1

**OGÓLNA CHARAKTERYSTYKA APLIKACJI PRZETWARZAJĄCEJ STRUMIENIE ZDARZEŃ**

## Wprowadzenie

Aplikacja przetwarzająca strumienie danych jest nieczęsto spotykanym rodzajem oprogramowania. Aby powstała musi zaistnieć szczególna potrzeba biznesowa, a taka rodzi się zwykle w dużych firmach, które widzą konieczność w przechowywaniu i przetwarzaniu olbrzymich ilości danych. Do tej kategorii zaliczają się firmy produkujące, czy obsługujące farmy wiatrowe. Każda turbina oprócz widocznych z zewnątrz części składowych, zawiera też pewną ilość komputerów i działającego na nich oprogramowania. Standardowym rozwiązaniem tego typu są systemy SCADA[[1]](#footnote-1). W każdej sekundzie, czy minucie odczytują one różnorakie parametry turbin, takie jak wymienione we wprowadzeniu dane telemetryczne typu temperatury skrzyni biegów, naprężeń istotnych elementów mechanicznych, czy wyprodukowanej mocy. Aby z tych danych skorzystać, należy je przesłać w miejsce zdolne obsłużyć duże ich ilości. Na ogół będą to nierelacyjne bazy danych, lub magazyny danych zbudowane w oparciu o chmurę – oba te rozwiązania brane są pod uwagę z powodu łatwego horyzontalnego skalowania. Kiedy do naszego systemu dochodzą dane z kolejnych turbin, chcemy w łatwy sposób umożliwić ich obsługę, najlepiej jeśli działoby się to jak najbardziej automatycznie, a wymienione rozwiązania to właśnie zapewniają.

Oprogramowanie tego typu można wytwarzać w dowolnym języku kompilowanym. Języki skryptowe nie powinny raczej być brane pod uwagę do tworzenia rdzenia tego typu systemu z uwagi na ich powolność. Mogą jednak pełnić pewną pomocniczą rolę. Przykładem takiego języka może być PowerShell, który w niniejszym projekcie użyty został do zarządzania konfiguracjami kontenerów dla lokalnie prowadzonego developmentu.

## C# i .Net

C# jest obiektowym językiem ogólnego przeznaczenia. Sprawdza się w wielu zastosowaniach – od programowania serwerowego w aplikacjach przetwarzających zarówno małe jak i duże ilości danych opartych o ASP.NET Core[[2]](#footnote-2), przez tworzenie przeglądarkowych frontendów opartych na silniku Blazor[[3]](#footnote-3), czy aplikacji mobilnych opartych o środowisko Xamarin[[4]](#footnote-4), aż do gier pisanych w oparciu o framework Unity[[5]](#footnote-5). Historycznie bazował na Javie i faktycznie składnia jego pierwszych wersji bardzo mocno ją przypominała. Jego wielkim ograniczeniem było też to, że oprogramowanie w nim napisane można było wytwarzać i uruchamiać jedynie pod systemem operacyjnym Windows. Z czasem, według opinii społeczności, prześcignął swój pierwowzór w zakresie wygody programowania i szybko wdrażanych, przydatnych usprawnień, a z wejściem na rynek środowiska uruchomieniowego .Net Core, język C#, a także inne języki uruchamiane na tej platformie (VB, F#), stał się wieloplatformowy i obecnie bez problemu napisane w nim oprogramowanie można uruchamiać zarówno na Windowsie, jak i systemach Linux, czy Unix.

Środowisko .Net z kolei jest tym wirtualną maszyną zapewniającą programom pisanym przy użyciu C# odpowiednie abstrakcje, zarządzanie i automatyczne odśmiecanie pamięci [[6]](#footnote-6)oraz inne mechanizmy, których nie może zabraknąć w nowoczesnej platformie programistycznej.

# Rozdział 2

**WYKORZYSTANE NARZĘDZIA I TECHNOLOGIE**



## Rider

IDE (Integrated Development Environment) Rider [[7]](#footnote-7)jest konkurencyjnym względem Visual Studio środowiskiem programistycznym. Jego twórcą jest znana na rynku firma JetBrains, która specjalizuje się w wytwarzaniu tego typu oprogramowania. Inne ich produkty, to np. IDE szeroko znane w społeczności programistów Python – PyCharm, czy w środowisku data scientist’ów – DataSpell. Wybrane zostało z uwagi na dużo większą responsywność i szybkość działania względem domyślnego w środowisku programistów C# IDE jakim jest Visual Studio.

## DataGrip

Jest to kolejne IDE pomocne w wytwarzaniu oprogramowania integrującego się z mniej popularnymi, a jednocześnie bardziej specjalistycznymi bazami danych typu MongoDB, czy użyta tutaj Cassandra.

## Visual Studio Code

Jest to bardzo lekkie, proste środowisko programistyczne ułatwiające pracę z językami skryptowymi, takimi jak Powershell.

## Apache Cassandra[[8]](#footnote-8)

Jest to rozproszony system zarządzania bazą danych o charakterze open source. Został zaprojektowany do obsługi dużej ilości rozproszonych danych na wielu serwerach, który będzie nadal funkcjonował nawet jeśli jeden z serwerów przestanie działać. Baza danych została nazwana po wieszczce Kasandrze z mitologii greckiej. Jest to rozwiązanie typu NoSQL, które zostało pierwotnie stworzone przez Facebooka. Udostępnione jako open source w 2008. W styczniu 2009 Facebook przekazał ją do Apache Software Foundation, pełnoprawnym projektem została rok później.

W niniejszym projekcie traktowana jest jako główny magazyn, z którego w razie potrzeby można będzie odtworzyć pomocnicze agregaty rezydujące w ElasticSearch.

## ElasticSearch

Jest to oprogramowanie komputerowe służące do wyszukiwania informacji stworzone przez Shaya Banona z firmy Elastic NV w roku 2010. Jako główny silnik wyszukiwania, system Elasticsearch[[9]](#footnote-9) wykorzystuje bibliotekę Apache Lucene[[10]](#footnote-10). Z tego m.in. powodu system Elasticsearch jest często porównywany do platformy o podobnych funkcjach i zastosowaniach Apache Solr. Wiele elementów systemu Elasticsearch jest dostępnych w postaci otwartych źródeł na serwisie GitHub. Według serwisu DB-engines, Elasticsearch jest najpopularniejszym silnikiem wyszukiwania na świecie.

W niniejszym projekcie ElasticSearch służy jako magazyn pomocniczy, przechowujący, oraz ułatwiający wyszukiwanie zagregowanych z dzienną rozdzielczością danych pochodzących z turbin wiatrowych.

## Kafka

Jest to broker wiadomości dostępny jako otwarte oprogramowanie. Projekt jest napisany w Scali, a rozwija go Apache Software Foundation. Cel przedsięwzięcia to umożliwienie obsługi danych czasu rzeczywistego pochodzących z wielu węzłów. Ujednolicona obsługa strumieni wejściowych ma zapewniać dużą przepustowość i redukcję opóźnień. Projekt powstał pod silnym wpływem dzienników transakcji.

Apache Kafka początkowo zaprojektowano w LinkedIn, a następnie udostępniono jako oprogramowanie opensource na początku 2011 roku. Projekt opuścił Apache Incubator 23 października 2012 roku. W listopadzie 2014 roku kilku inżynierów, którzy pracowali nad Kafką w LinkedIn, założyło nowe, silnie związane z Kafką przedsiębiorstwo o nazwie Confluent.

# Rozdział 3

**PROJEKT APLIKACJI**



## Struktura aplikacji

Projekt składa się z dwóch głównych części – konsolowej aplikacji emulującej wysyłanie danych w taki sposób, jakby robiła to prawdziwa turbina wiatrowa oraz kolejnej konsolowej aplikacji, odczytującej i przetwarzającej te dane.

Aplikacja ta nie zawiera żadnych widoków, ponieważ jest typowo serwerowym oprogramowaniem, działającym głęboko w infrastrukturze firmy. Jej jedynym zadaniem, jak zostało wspomniane, jest umieszczenie danych w poprawnej formie, w dwóch magazynach oraz logowanie postępu importu.

## Wymagania techniczne

Aplikacja ta została przygotowana do używania na urządzeniach typu PC (w tym, przede wszystkim wszelkiej klasy środowiska serwerowe) i laptop. Z uwagi na wysokie wymagania dotyczące ilości rdzeni procesora i dostępnej pamięci, tylko te dwa środowiska zdolne będą ją uruchomić.

Jeśli chodzi o wymagania sprzętowe stawiane środowisku lokalnemu, takiemu jak laptop, są to:

* system operacyjny zdolny uruchamiać skrypty Powershell – Windows lub Linux z zainstalowanymi stosownymi pakietami
* zainstalowane oprogramowanie Docker Desktop
* procesor zdolny uruchomić 5 wątków
* 14gb pamięci RAM

Są to wymagania minimalne, takie które sprawią, że uruchomienie tej aplikacji nie sprawi, że komputer przestanie być responsywny.

Jeśli chodzi o wymagania sprzętowe stawiane środowisku serwerowemu, są to:

* system operacyjny Linux na serwerze uruchamiającym aplikację
* co najmniej trzy niezależne serwery hostujące węzły bazy danych Cassandra (4-rdzeniowe procesory, co najmniej 128gb pamięci RAM)
* co najmniej trzy niezależne serwery hostujące węzły bazy danych ElasticSearch (technicznie, jw.)

# Rozdział 4

**IMPLEMENTACJA APLIKACJI**



## Implementacja

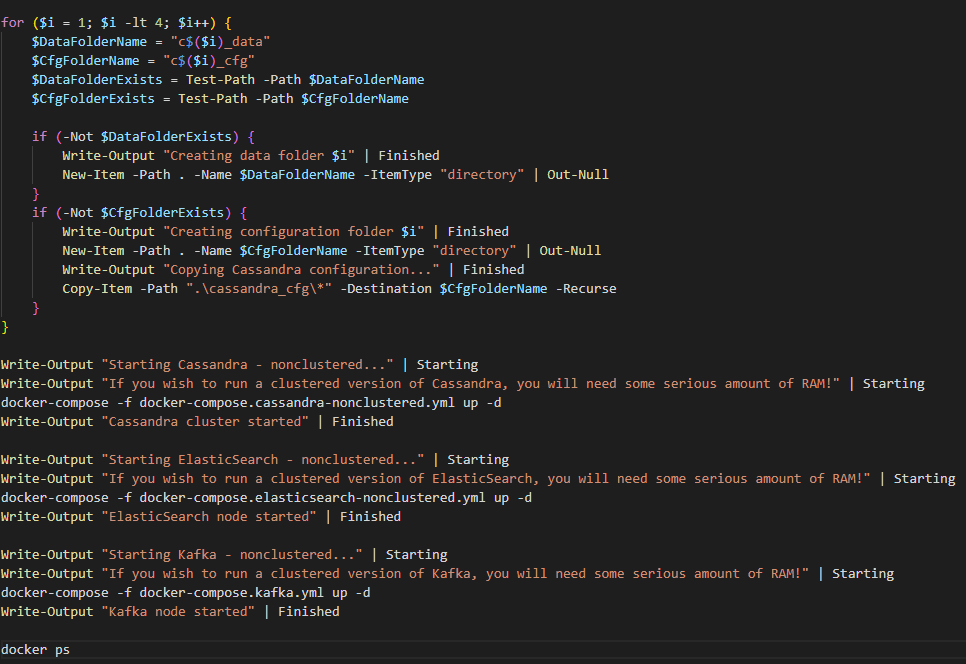
Tworzenie aplikacji tego typu należy rozpocząć od instalacji niezbędnych komponentów infrastrukturalnych. Pierwszym tego typu elementem jest oprogramowanie Docker Desktop. Dla systemu Windows jego autorzy przewidzieli dedykowany instalator, który należy ściągnąć z internetu i uruchomić. Podczas instalacji należy zaakceptować domyślnie wybrane opcje.

Po zainstalowaniu Dockera należy uruchomić w nim bazy danych wymagane przez aplikację do działania. W rzeczywistości, kiedy aplikacja uruchomiona byłaby na serwerze, Docker wymagany byłby tylko po to, aby móc ją uruchamiać, natomiast same bazy danych uruchamiane byłyby dla zapewnienia najwyższej wydajności w trybie tzw. „bare metal” – bezpośrednio na serwerze.

Projekt zawiera skrypt automatyzujący instalację wymaganego oprogramowania. Znajduje się on w ścieżce (zaczynając od głównego folderu projektu): /devops/runme.ps1.

Rysunek 1

Skrypt automatyzujący infrastrukturę

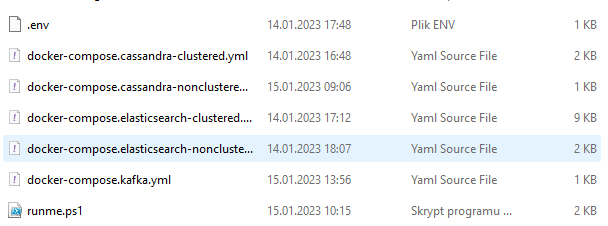


Źródło: opracowanie własne

Jego jedyną rolą jest uruchomienie oprogramowania docker, które automatycznie zainstaluje elementy zdefiniowane w plikach docker compose widocznych poniżej:

Rysunek 2

Lista plików docker-compose

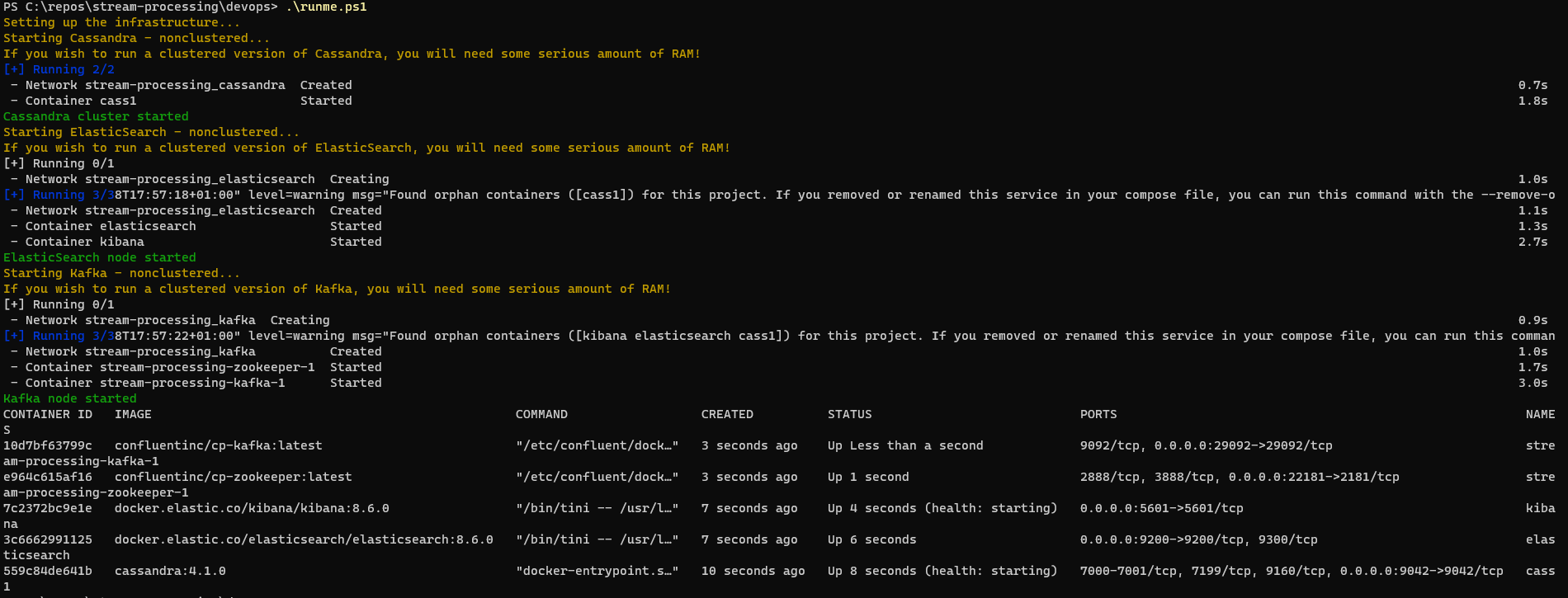


Źródło: opracowanie własne

Powszechną praktyką, w wypadku konfiguracji skomplikowanej infrastruktury przy użycia Dockera jest użycie oddzielnych plików definiujących obrazy[[11]](#footnote-11) dla ułatwienia zarządzania nimi. Po zakończeniu pracy skryptu, w konsoli powinniśmy zobaczyć rezultat podobny do poniższego:

Rysunek 3

Rezultat uruchomienia skryptu runme.ps1

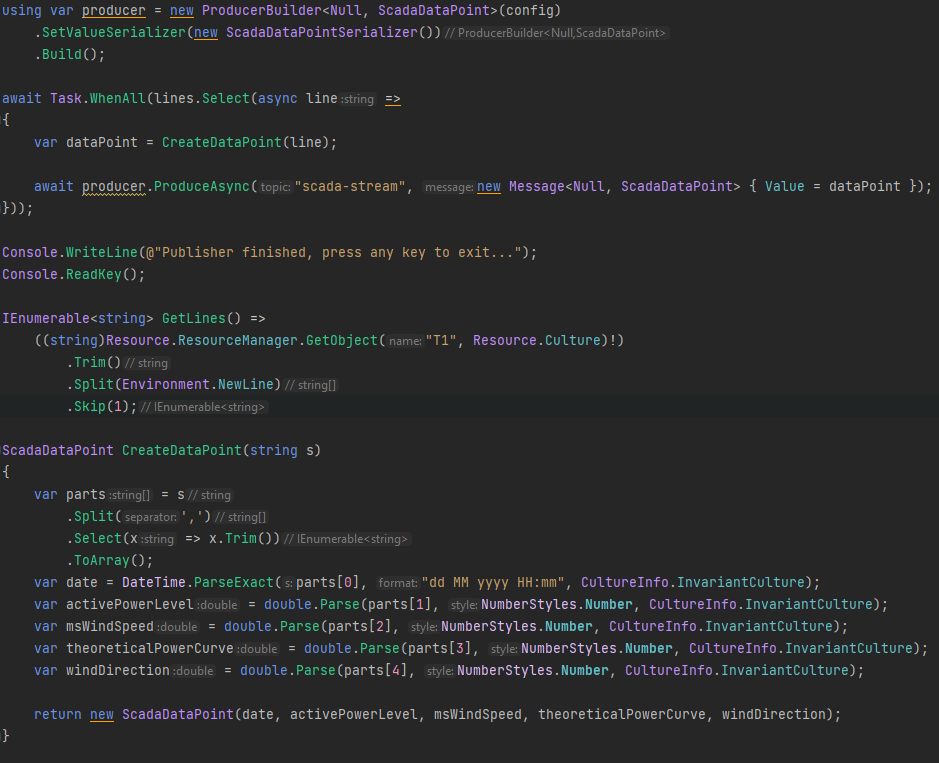


Źródło: opracowanie własne

Kolejnym krokiem jest stworzenie aplikacji umieszającej dane w kolejce Kafka. Potrzebujemy jej z uwagi na testowanie – tworząc aplikację nie mamy możliwości interakcji z prawdziwym systemem SCADA. Nie mamy jej aż do momentu wejścia na środowisko preprodukcyjne, chociaż równie często spotykana sytuacja wygląda tak, że nie ma się styczności z faktycznymi turbinami aż do wejścia na produkcję. Stąd, aby móc przetestować cały proces, musimy jego początek emulować. Nie jest to trudne, a mając dobrze zdefiniowane wymagania biznesowe dotyczące tego, co mamy składować w bazach danych kod takiego emulatora (po zignorowaniu importu bibliotek i początkowych linii konfiguracyjnych) wygląda następująco:

Rysunek 4

Kod programu - producenta danych



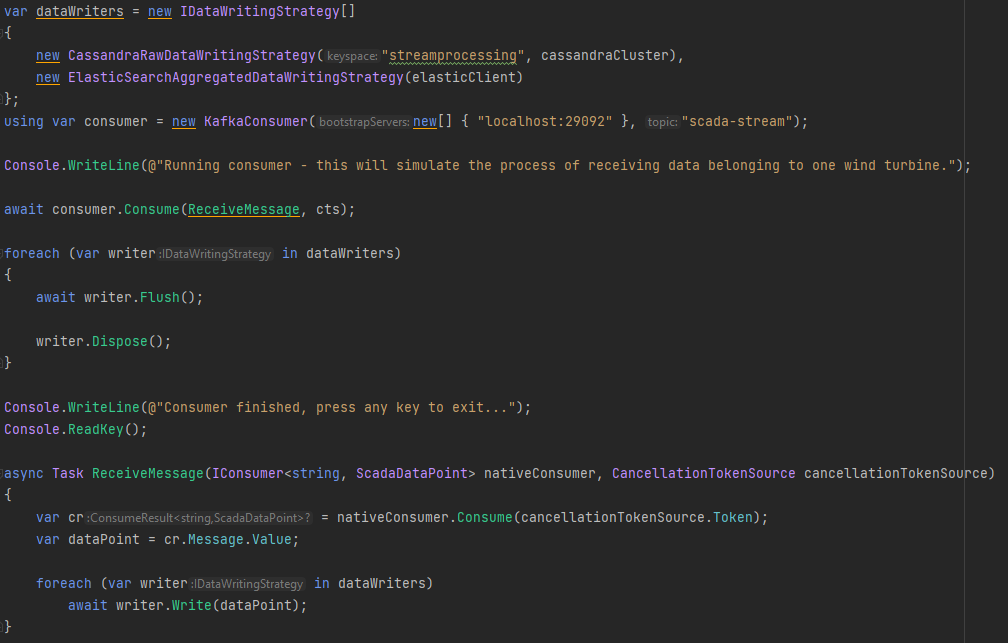
Źródło: opracowanie własne

Pierwsze linie widoczne na powyższym rysunku starają się równolegle wysłać wiele wierszy danych. Odczyt owych linii (z pliku) definiuje funkcja GetLines, następnie obiekt typu ScadaDataPoint tworzony jest z kolejnych linii przy użyciu metody CreateDataPoint.

Jak zostało wspomniane wcześniej, program ten jest tylko emulatorem – pomocą developerską istniejącą po to, aby lokalny rozwój aplikacji był możliwy i sam nie będzie umieszczany na produkcji. Dużo ciekawszym zatem programem jest aplikacja konsumenta. Kod jej punktu wejścia zaprezentowany został na kolejnym rysunku:

Rysunek 5

Punkt wejściowy aplikacji - konsumenta



Źródło: opracowanie własne

Pierwsze linie definiują dwie strategie[[12]](#footnote-12) zapisu danych. Aby wykorzystać polimorfizm[[13]](#footnote-13), a więc uprościć kod programu i uczynić go gotowym na ewentualne, przyszłe zmiany, a jednocześnie bardziej ogólnym, wzmiankowane strategie implementują interfejs IDataWritingStrategy. W tej prototypowej aplikacji powstały jedynie dwie strategie, ale z tak napisanym kodem, łatwo można rozszerzyć ich listę o kolejne.

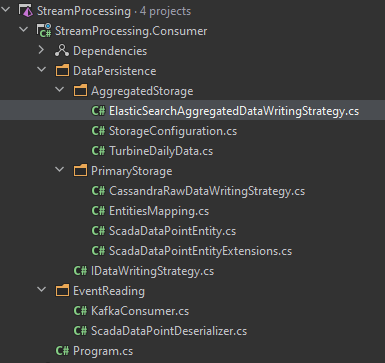
Pierwszą z nich jest CassandraRawDataWritingStrategy. Ma ona na celu zapis surowych danych – w niezmienionej formie. Powodem jej istnienia jest to, że nasza firma chce, aby jej analitycy mogli z takich surowych danych budować własne modele dynamicznie. Innymi słowy dzisiaj dla firmy może najbardziej istotny będzie poziom produkowanej mocy, a więc wszystkie analizy powinny być prowadzone na tej tylko kolumnie. W przyszłości jednak wiemy, że istotne będzie też badanie awaryjności turbin wiatrowych. Oprócz poziomu mocy istnieją też inne wskaźniki, które mogą analitykom i ich modelom uczenia maszynowego sugerować, że dla danej turbiny istnieje pewne wysokie prawdopodobieństwo nadchodzącej awarii.

Drugą zaimplementowaną strategią jest ElasticSearchAggregateDataWritingStrategy. Ta ma na celu budowanie dziennych agregatów danych. Oczywiście analitycy mogliby je sami budować, jednak tutaj cel jest nieco inny. Wymaganie biznesowe jest takie, aby aplikacja frontendowa potrafiła generować pewne typy prostych raportów na żądanie, bez udziału analityków. Aby temu wymaganiu sprostać, powstał pomysł dziennych agregacji danych.

Prócz tych dwóch, kluczowych elementów aplikacji konsumenckiej, zawiera ona kolejny – KafkaConsumer. Nie zapisuje on danych, lecz odczytuje je z kolejki Kafka i przekazuje dwóm opisywanym wcześniej strategiom. Aby podsumować kształt aplikacji konsumenta, załączony zostaje kolejny rysunek:

Rysunek 6

Wysokopoziomowy rzut na elementy aplikacji



Źródło: opracowanie własne

## Testowanie

Podczas testowania zostały przeprowadzone:

* Testy funkcjonalne
  + Sprawdzenie poprawności wysyłki danych
  + Sprawdzenie poprawności składowania danych w obu bazach
* Testy wydajnościowe
  + Sprawdzenie wydajności aplikacji pod kątem wolumenu danych, który zdolna jest obsłużyć

Dzięki poprawnemu użyciu technologii przeznaczonych dokładnie dla opisywanego scenariusza udowodniono, że cały wybrany stos technologiczny będzie skalował się liniowo wraz z dodawaniem do systemu kolejnych danych.

## Wdrożenie

Sugerowanych jest tutaj kilka rozwiązań. Aplikacja została przygotowana w taki sposób, aby mogła operować zarówno w środowisku „on-premise[[14]](#footnote-14)”, jak i chmurowym. Poza infrastrukturą w postaci wymienionych już bazy danych Cassandra, ElasticSearch oraz systemu wysyłki zdarzeń Kafka, aplikacja nie ma zewnętrznych zależności. Zatem, jeśli poda się jej odpowiednie dane konfiguracyjne takie jak hasła dostępowe i adresy serwerów, bez zmian samego kodu możliwe będzie jej przeniesienie na dowolny serwer, komunikujący się z dowolną ilością serwerów hostujących wymienione części infrastruktury.

# BIBLIOGRAFIA

## Opracowania książkowe

1. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: **Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku.** – Wydawnictwo Helion
2. Jeff Carpenter, Eben Hewitt: **Cassandra: The Definitive Guide, 3rd Edition.** – Wydawnictwo O’Reilly
3. Clinton Gormley: **Elasticsearch: The Definitive Guide**. – Wydawnictwo O’Reilly
4. Neha Narkhede, Gwen Shapira, Todd Palino: **Kafka: The Definitive Guide.** – Wydawnictwo O’Reilly

## Netografia

1. https://en.wikipedia.org/wiki/SCADA (data odczytu 17.01.2023)
2. https://learn.microsoft.com/pl-pl/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-7.0 (data odczytu 17.01.2023)
3. https://learn.microsoft.com/pl-pl/aspnet/core/blazor/?view=aspnetcore-7.0 (data odczytu 17.01.2023)
4. https://learn.microsoft.com/pl-pl/xamarin/ (data odczytu 17.01.2023)
5. https://unity.com/how-to/learning-c-sharp-unity-beginners (data odczytu 17.01.2023)
6. https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals (data odczytu 17.01.2023)
7. https://www.jetbrains.com/rider/ (data odczytu 17.01.2023)
8. https://cassandra.apache.org/\_/index.html (data odczytu 17.01.2023)
9. https://pl.wikipedia.org/wiki/Elasticsearch (data odczytu 17.01.2023)
10. https://lucene.apache.org/ (data odczytu 17.01.2023)
11. https://blog.packagecloud.io/what-is-a-docker-image/ (data odczytu 18.01.2023)
12. https://refactoring.guru/design-patterns/strategy (data odczytu: 18.01.2023)
13. https://www.modestprogrammer.pl/co-to-jest-polimorfizm-w-programowaniu-obiektowym (data odczytu: 18.01.2023)
14. https://www.forbes.com/sites/forbestechcouncil/2019/03/22/cloud-based-vs-on-premise-servers/?sh=a86be3f79e20 (data odczytu: 18.01.2023)

# SPIS RYSUNKÓW

Rysunek 1 9

Rysunek 2 10

Rysunek 3 10

Rysunek 4 11

Rysunek 5 12

Rysunek 6 13

1. https://en.wikipedia.org/wiki/SCADA (data odczytu 17.01.2023) [↑](#footnote-ref-1)
2. https://learn.microsoft.com/pl-pl/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-7.0 (data odczytu 17.01.2023) [↑](#footnote-ref-2)
3. https://learn.microsoft.com/pl-pl/aspnet/core/blazor/?view=aspnetcore-7.0 (data odczytu 17.01.2023) [↑](#footnote-ref-3)
4. https://learn.microsoft.com/pl-pl/xamarin/ (data odczytu 17.01.2023) [↑](#footnote-ref-4)
5. https://unity.com/how-to/learning-c-sharp-unity-beginners (data odczytu 17.01.2023) [↑](#footnote-ref-5)
6. https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals (data odczytu 17.01.2023) [↑](#footnote-ref-6)
7. https://www.jetbrains.com/rider/ (data odczytu 17.01.2023) [↑](#footnote-ref-7)
8. https://cassandra.apache.org/\_/index.html (data odczytu 17.01.2023) [↑](#footnote-ref-8)
9. https://pl.wikipedia.org/wiki/Elasticsearch (data odczytu 17.01.2023) [↑](#footnote-ref-9)
10. https://lucene.apache.org/ (data odczytu 17.01.2023) [↑](#footnote-ref-10)
11. https://blog.packagecloud.io/what-is-a-docker-image/ (data odczytu 18.01.2023) [↑](#footnote-ref-11)
12. https://refactoring.guru/design-patterns/strategy (data odczytu: 18.01.2023) [↑](#footnote-ref-12)
13. https://www.modestprogrammer.pl/co-to-jest-polimorfizm-w-programowaniu-obiektowym (data odczytu: 18.01.2023) [↑](#footnote-ref-13)
14. https://www.forbes.com/sites/forbestechcouncil/2019/03/22/cloud-based-vs-on-premise-servers/?sh=a86be3f79e20 (data odczytu: 18.01.2023) [↑](#footnote-ref-14)